In The Heart Of Applesoft

This article is not written to know how Applesoft works, [illegible] to work wit[h] [illegible], or more specifically [illegible] other to us[e] [illegible]

[illegible] 2 [illegible] program [illegible]. This program [illegible] 5 times faster (depending on the numbers [of] the matrices) than a comparable BASIC program.

C. Bongers
Erasmus University
Postbus 1738
3000 DR Rotterdam
The Netherlands

[My] primary motivation [t]o buy a microcomputer was to develop a number of statistical programs which were to be used for a research project I [wa]s working on. After comparing [illegible] micros with each other, with [illegible] execution speed of BASIC [illegible] possibilities, [illegible] The [illegible] delivered [illegible] BASIC [i]n ROM. After

[illegible] the [illegible] of the [illegible] are all the [illegible] given sequence of symbols (for instance, ABC has the permutations [AB]C, ACB, BAC, BCA, CAB, CBA) and [a] program to solve the 10 by 6 pentomino puzzle (see *BYTE*, Nov. 1979). The permutation program ran reasonably fast ([illegible] permutations per [se]cond) but the pentomino program turned out to be a disappointment. [Af]ter waiting for several hours, it finally produced the first of the 2,339 solutions, so I never bothered trying to find more solutions.

Example [illegible]							
	[illegible]			[M]antissa			Sign
MFP	[illegible]	\$9[illegible]	[illegible]	\$AC	[illegible]	[illegible]	\$A[illegible]
[illegible] contents		[illegible]	[illegible]		[illegible]	[illegible]	
Decimal		[illegible] × $\frac{1}{16}$ +	3 × $\frac{1}{256}$ +	11 × $\frac{1}{4096}$ +	0 × [illegible] +	12 × $\frac{1}{16^{[illegible]}}$	[illegible]90380859

[T]able [illegible]: Applesoft [R]outines

I. General remarks

A. Notation

1. (→ A,Y) means: pointed to by Accumulator (low) and Y register (high).

2. (A = Z.F. = \$X) means: Accumulator has to contain, or contains the contents of location \$X. If \$X equals zero, the zero flag (Z.F.) is set or must be set, otherwise the zero flag is or must be clear.

B. Remarks

1. For some routines presented below, the entry and/or exit values of the Accumulator, the X register and the Y register are given. If no entry is specified, no entry is necessary. If no exit or not all registers [o]n exit are specified, the registers not specified may have unpredictable values after the execution of the Applesoft routine.

2. For each routine, the memory locations that may be modified by the (error free) execution of the routine are given.

C. Warning

1. When working with m.l. programs that are called from BASIC, one may wish to use zero page locations to store temporary results. However, a number of zero page locations are initialized to certain values at the cold/warm start of Applesoft and changing the contents of these locations may lead to unexpected results. Furthermore there are a number of locations in which Applesoft stores information during the execution of the BASIC program, such as the current line number or the pointer to the line from which data is being read. Clobbering one of these locations usually has the effect that the program will crash sooner or later.

In order to avoid problems when working with zero page addresses it is therefore recommended to consult the zero page usage map in the Applesoft manual first. (See also the memory atlas constructed by Prof. W. F. Luebbert, published in the August 1979 issue of MICRO.)

(continued)

At that time, however, I discovered that the Apple can also be programmed rather easily in machine language with the help of the mini-assembler. Since I was interested to know what speed gain could be obtained, I translated the permutation program in machine code. To my surprise, the program ran about 675 times faster (approximately 13,500 permutations per second) than the BASIC permutation program. Of course, I immediately got my pentomino program and translated this in machine code too. The 2,339 solutions now came out in less than 3 hours, which means also a considerable gain in speed as compared to the BASIC program.

When to Use Machine Language Programs or Subroutines

Some programs, like those mentioned above, can easily be translated from BASIC to machine code. However, for the majority of the programs that I intend to write, this is not the case, since in these programs floating point variables rather than "one byte" variables have to be used. For some floating point arithmetic such as addition and multiplication, it is probably possible to write the routines yourself, but for functions such as the sine and the logarithm this would mean a lot of work. Furthermore, being busy with "trying to reinvent the wheel" is not a very stimulating idea.

However, there is a fairly easy way out of this problem. All the routines needed for floating point arithmetic, have to be somewhere in the Applesoft ROM, so all one has to do is list Applesoft and try to understand how it works. After locating the entries of the floating point routines, these routines then can be called by the machine language (m.l.) program. Although the whole process can be written down in a few lines, [illegible] several weeks of hard work before I knew enough of Applesoft to write, as an exercise, a matrix multiplication subroutine in m.l. which can be called from BASIC by means of the & symbol. This program runs about 8 times faster than a BASIC matrix multiplication subroutine and further has the advantage that the names of the matrices can be passed in an easy way. On the other hand, a disadvantage is that the m.l. program uses more memory space than the BASIC program. At the end of this article, the matrix multiplication program will be more extensively discussed.

Table 1: Applesoft Routines (continued)

Of course, this warning does not apply to (most of the) zero page locations that may be modified by the routines described below. For instance, if one uses neither the power function nor SQR nor trigonometrical functions, it will be safe to use locations $8A-$8E, since these locations are used by none of the other functions (routines) listed in this table.

2. If neither strings nor high-resolution graphics nor ON ERR statements are used, one can (probably) safely store temporary results in the following zero page locations.

   $6-$9, $17-$1F, $58-$5D, $71-$72, $CE-$D5, $D7, $D9-$EF, $F4-$FF

II. Description and entries of the routines

A. Charget-Charcheck

1. Purpose

   The memory locations $B8 and $B9 contain—during the execution of a BASIC program—a text pointer which points to the last retrieved character of the BASIC program. The Charget routines can be used to load the next character or the current character (again) in the Accumulator. To determine whether the character equals a predetermined symbol one of the Charcheck routines may be used.

2. Charget routines

   $B1: Advance text pointer and load next character in the Accumulator (spaces are ignored).

   Exit(A = next character, X = entry, Y = entry).

   Exit Status: Carry is clear if character is a digit (hex value:30-39), otherwise carry is set. Zero flag is set if character equals 0 (= end of line sign) or 3A (= end of statement sign, i.e. ":"), otherwise zero flag is cleared.

   Modifies $B8,$B9.

   $B7: Load current character another time in the Accumulator.

   Exit(A = current character, X = entry, Y = entry).

   For status see subroutine $B1.

3. Charcheck routines

   $E07D: Check whether character in Accumulator is a letter.

   Entry(A), Exit(A = entry, X = entry, Y = entry).

   Exit Status: Carry is set if character is a letter, otherwise carry is cleared.

   The following 4 routines can be used to check whether the text pointer points to a specific symbol. If the result of the check is positive, the next character is loaded in the Accumulator by means of the execution of subroutine $B1. In the other case, the message "SYNTAX ERROR" is displayed and Applesoft returns to BASIC command level. The exits of the 4 routines are:

   Exits(A = next character, X = entry, Y = 0), modify $B8,$B9.

   $DEC0: Check whether the character that is pointed to by the text pointer equals the character in the Accumulator.

   Entry (A).

   $DEB8: Check whether the text pointer points to a right parenthesis.

   $DEBB: Check whether the text pointer points to a left parenthesis.

   $DEBE: Check whether the text pointer points to a comma.

B. Compare

1. Purpose

The compare routines can be used for comparing a real variable in the MFP with a real variable in the SFP or a real variable in memory.

2. Compare routines

$DF6A: Compare MFP with SFP according to the status of the comparison in location $16. The result of the comparison (1 if true, 0 if false) is converted to a real variable in the MFP. The various types of comparisons are listed below.

Type of Comparison	$16 has to be put equal to:	Result comparison
>	1	1 if SFP > MFP, else 0
=	2	1 if SFP = MFP, else 0
<	4	1 if SFP < MFP, else 0
> =	3	1 if SFP ≥ MFP, else 0
< >	5	1 if SFP ≠ MFP, else 0
< =	6	1 if SFP ≤ MFP, else 0

Modifies $60,$61,MFP,SFP.

$EBB2: Compare MFP with memory (→ A,Y).

Entry(A,Y), Exit(A = FF if MFP < memory, A = 0 if MFP = memory, A = 1 if MFP > memory), modifies $60,$61.

C. Conversion

1. Purpose

The Conversion routines can be used to convert:

a) a real in the MFP to an integer
b) a one or two byte integer to a real in the MFP

Unless specified otherwise, all integers are assumed to be two's complement integers.

2. Real to integer conversion routines

$EBF2: Convert MFP to integer. The number in the MFP must be between -2^{31} and 2^{31} (notation: $-2^{31} < MFP < 2^{31}$). Result is stored in mantissa of MFP (locations $9E-$A1).

Exit(Y = 0), modifies MFP.

$E752: Convert MFP, where $-2^{16} < MFP < 2^{16}$, to two byte integer. Store result in $50 (low) and $51 (high).

Exit(A = $51, Y = $50), modifies $50,$51,MFP.

Remark: "Wrap around" occurs if the absolute value of the number in the MFP is larger than $2^{15} - 1$.

$E10C: Convert MFP, where $-2^{15} < MFP < 2^{15}$, to two byte integer. Store result in $A0 (high) and $A1 (low).

Exit (Y = 0), modifies $60,$61,MFP.

$E108: Same as $E10C, except that entry-value of MFP must be: $0 \leq MFP < 2^{15}$.

$DA65: Pack extension byte in MFP and convert MFP, where $-2^{15} < MFP < 2^{15}$, to two byte integer. Store integer (high byte first) in (→ $85,$86).

Exit(Y = 1), modifies $60,$61,MFP.

(continued)

An important point to note is that, as a consequence of using floating point arithmetic, there is a significant drop of the speed gain, namely from a factor 675 obtained with the permutation program to a factor 8 obtained with the matrix multiplication program. The reason is that—when multiplying matrices—a relatively large portion of the CPU time is used for the multiplication and addition of floating point numbers. Whether this is done under control of a BASIC program, or by calling the appropriate routines in Applesoft from a m.l. program, makes no difference, since in both cases the same multiplication and addition routines are used. The gain of speed that occurs in the m.l. matrix multiplication program is obtained by short-cutting the time-consuming determination of the pointers to array elements in BASIC.

It will now also be clear that it does not make any sense to calculate for instance, 1000 logarithms by means of a m.l. program. When written in BASIC, thus

```
10 FOR I = 1 TO 1000 :
   A = LOG (I) : NEXT
```

the program will run approximately 23 seconds. About 90% of this time, the computer will be busy with the calculation of the logarithms, and about 10% of the time with the parsing of the statements and the evaluation of the FOR...NEXT loop. When writing a m.l. program to calculate the logarithms, one may expect it to run no more than 10% faster than the BASIC program, since as to the calculation of the logarithms, no time can be saved.

Therefore, with respect to gaining speed, it is only profitable to write a m.l. program or subroutine if, in this way, time-consuming access to array elements can be short-cutted or iterative parts of the program can be made more efficient. Some examples where m.l. routines will be useful are: finding the largest element of an array, calculating the inverse of a matrix, sorting the elements of a vector, or calculating probabilities under a bivariate (log) normal distribution.

Apart from gaining speed, there may however be other arguments for writing m.l. routines. For instance, one may wish to extend tape or disk versions of Applesoft with some self-written BASIC commands or functions. Also, it can be attractive to make frequently used subroutines more independent of

Table 1: Applesoft Routines (continued)

3. Integer to real conversion routines

$E2F2: Convert two byte integer in A (high) and Y (low) to real in MFP.

Entry(A,Y), Exit(Y = 0), modifies MFP, puts $11 equal to zero.

$E301: Convert one byte integer in Y to positive real in MFP. (The integer in Y is thus not interpreted as a two's complement integer.)

Entry(Y), Exit(Y = 0), modifies MFP, puts $11 equal to zero.

$EB93: Convert one byte integer in Accumulator to real in MFP.

Entry(A), Exit(Y = 0), modifies MFP.

$DEE9: Pull integer (%) variable from memory (→$A0,$A1) into A (high) and Y (low). Next, convert integer to real in MFP.

Exit(Y = 0), modifies MFP, puts $11 equal to zero.

D. Copy

1. Purpose

The Copy routines can be used to

a) pull data (from memory) into the MFP or the SFP
b) pack the MFP and store the MFP in memory
c) copy the MFP into the SFP and vice versa
d) push the MFP on stack or pull the SFP from stack

The Copy routines are for real variables only. For routines that handle integer (%) variables see Conversion.

2. MFP routines

$EAF9: pull memory (→ A,Y) into the MFP and put the extension byte equal to zero.

Entry(A,Y), Exit(A = Z.F. = $9D,X = entry,Y = 0), modifies $5E,$5F,MFP.

$EAFD: Pull memory (→ $5E,$5F) into the MFP and put the extension byte equal to zero

Exit(A = Z.F. = $9D,X = entry,Y = 0), modifies MFP.

$DE10: Pack extension byte in MFP and push MFP on stack (6 bytes).

Exit(A = Z.F. = $9D), modifies $5E,$5F,MFP.

The following four routines pack the sign and the extension byte in the MFP, store the MFP in the locations indicated and put the extension byte equal to zero.

For these four routines the exits are:

Exits(A = Z.F. = $9D,Y = 0), modify $5E,$5F, MFP.

$EB1E: store MFP in $98-$9C

$EB21: store MFP in $93-$97

$EB27: store MFP in (→ $85,$86)

$EB2B: store MFP in (→ X,Y)

3. SFP routines

$E9E3: Pull memory (→ A,Y) in the SFP and determine $AB (= the exclusive OR of the signs of the numbers in the MFP and the SFP).

Entry(A,Y), Exit (A = Z.F. = $9D,X = entry,Y = 0), modifies $5E,5F,SFP,$AB.

$E9E7: Pull memory (→ $5E,$5F) in the SFP and determine $AB.

Exit(A = Z.F. = $9D,X = entry,Y = 0), modifies SFP,$AB.

(continued)

the main program, so that parameters can be passed by value rather than by name, which in BASIC is only possible by means of a lot of PEEKs and POKEs. Last but not least, one may like the challenge involved in writing m.l. programs.

The Main and Secondary Floating Point Accumulator

Before presenting the Applesoft routines that can be of help when writing m.l. programs, the main and secondary floating point accumulator, (henceforth to be abbreviated as MFP and SFP respectively), will shortly be discussed. Almost all the arithmetical and mathematical routines use the MFP and/or the SFP. The MFP occupies the memory locations $9D-$A2 and $AC. The exponent of the floating point number is in $9D (in excess 80 code), the mantissa is in $9E-$A1, and its sign is in $A2. Location $AC is used in most floating point routines as an extra mantissa byte, to increase the precision of the calculations. This location will further be called "the extension byte." An example of how one can convert the contents of the MFP to a decimal number is given in example 1 (page 31). The sign of the number is positive, since the first bit of $A2 is zero. In case this bit equals one, the sign of the number in the MFP will be negative. The exponent is calculated by converting the hex number 81 in $9D to decimal, which gives 129, and by subtracting the excess (= 80 (hex) or 128 (decimal)) from it. The method that is used to convert the mantissa to decimal is essentially the same as the method used to convert a normal hex number to decimal, except that instead of the multiplicands 16, 256, 4,096, etc., the reciprocals of these numbers have to be used.

The number zero forms an exception to the rules mentioned above. Applesoft considers a number to be zero if the exponent ($9D) equals zero, independent of the value of the mantissa.

The results from arithmetical operations and mathematical functions in Applesoft are, in general, placed in the MFP. Next, the MFP is usually normalized and pushed on the stack or stored in memory. The normalizing of the MFP means that the bytes of the mantissa are rotated to the left (zeros enter at the right) until the left-most bit of $9E equals one. At every rotation the exponent is decreased by one, since rotating the mantissa one bit to the left means multiplying the number in the MFP by two, and this number must, of course, remain the same.

If, after the normalizing process, the MFP has to be stored in memory, it must be packed because the MFP occupies 7 bytes of memory, whereas Applesoft reserves only 5 bytes for the storage of real variables. In the packing routine, first the mantissa is rounded off by considering the left-most bit of the extension byte. If this bit equals one, the mantissa is increased by one, otherwise the mantissa remains the same. Then the sign is packed into the floating point number. If the sign is positive, the left-most bit of $9E is put equal to zero, otherwise it remains equal to one. Note that the sign can be packed in this way because the first bit of $9E contains no information since it always equals one after normalizing.

The SFP occupies the memory locations $A5-$AA. The exponent is in $A5, the mantissa in $A6-$A9, and its sign in $AA. The SFP has no extension byte. For the arithmetical and mathematical operations requiring two operands, the first operand has to be put in the MFP and the second operand in the SFP. Thus, loading the SFP and the MFP with two numbers and doing a JSR to, for instance, the multiplication routine, leaves the product of the numbers in the MFP. For some arithmetical routines it is necessary to determine—before the routine is executed—the exclusive OR of the signs of the numbers in the MFP and the SFP. The result must be stored in location $AB. This implies that the first bit of $AB must be one if the signs differ, otherwise the first bit has to equal zero. However, in most cases the user does not have to bother about determining the value of $AB, since it usually is not necessary to load the MFP and/or the SFP "by hand." Applesoft provides us with a lot of routines that can be used to get floating point numbers from memory, unpack them, and place them in the MFP or the SFP. All the routines that pull memory in the SFP also set $AB to the right value.

The Use of Applesoft Routines

The Applesoft subroutines that are, in my opinion, the most useful for m.l. programmers are listed in table 1. A distinction has been made between various types of subroutines, such as Copy, Errors, Conversion and Mathematical routines, etc. Rather than discussing each of the routines separately, a (very) simple example will be given to illustrate how to work with them. For a good understanding of this example, it is advisable to read the general remarks in table 1 first.

Table 1: Applesoft Routines (continued)

$DE47: Pull stack in the SFP and determine $AB. This routine will usually be used in combination with subroutine $DE10. In that case it is for a successful execution of routine $DE47 necessary to push the return address of $DE47 on stack (high order byte first) before executing $DE10. Contrary to most other routines described here, $DE47 must be executed by means of a JMP instruction.

Exit(A = Z.P. = $9D, X = entry, Y = entry), modifies SFP,$AB.

4. SFP/MFP routines

$EB53: Copy SFP into MFP, put extension byte equal to zero.

Exit(A = $9D, X = 0, Y = entry), modifies MFP.

$EB63: Pack extension byte in MFP and copy MFP into SFP, put extension byte equal to zero.

Exit(A = $9D, X = 0, Y = entry), modifies MFP,SFP.

$EB66: Copy MFP (without extension byte) into SFP, put extension byte equal to zero.

Exit(A = $9D, X = 0, Y = entry), modifies MFP,SFP.

E. Errors

1. Purpose

If an error is detected in a m.l. program, one of the error routines may be used to print an error message.

2. Error messages

To print an error message, load the X register with the code of the message and execute a JMP to $D412 or execute a JMP to one of the locations listed behind the error messages. After printing the error message, Applesoft returns to BASIC command level (unless an ON ERR statement has been executed).

Code	Error message	JMP location
00	NEXT WITHOUT FOR	$DD08
10	SYNTAX ERROR	$DEC9
16	RETURN WITHOUT GOSUB	$D979
2A	OUT OF DATA	—
35	ILLEGAL QUANTITY	$E199
45	OVERFLOW	$E8D5
4D	OUT OF MEMORY	$D410
5A	UNDEF'D STATEMENT	$D97C
6B	BAD SUBSCRIPT	$E196
78	REDIM'D ARRAY	—
85	DIVISION BY ZERO	$EAE1
95	ILLEGAL DIRECT	$E30B
A3	TYPE MISMATCH	$DD76
B0	STRING TOO LONG	—
BF	FORMULA TOO COMPLEX	$E430
D2	CAN'T CONTINUE	—
E0	UNDEF'D FUNCTION	$E30E

F. Expressions

1. Purpose

The Expressions routines can be used to evaluate expressions in an & statement. When calling an expression evaluation routine, the text pointer in $B8 and $B9 must point to the first character of the expression. After control is returned from the evaluation routine, the text pointer points to the first character behind the expression. In the evaluation routines below, this character is called the terminal sign. The terminal sign might, for instance, be a comma, but also a special character such as a "#". The locations that are modified by the routines are not specified here, since these depend on the type of the expression.

2. Expression evaluation routines

$DD67: Evaluate expression to next terminal sign, store result in MFP.

$E105: Evaluate expression to next terminal sign and convert result, which must be non-negative, to two byte integer in $A0 (high) and $A1 (low).

Exit(Y = 0).

$E6F8: Evaluate expression to next terminal sign and convert result, which must be non-negative, to a one byte integer in $A1.

Exit(A = terminal sign, X = $A1, Y = 0).

G. Init

1. Purpose

Initialize mantissa of the MFP or the SFP.

2. Initialization routines

$EC40: Init mantissa MFP (except extension byte) and Y to value in Accumulator

Entry(A), Exit(A = entry, X = entry, Y = A), modifies MFP.

$E84E: Put MFP ($A2 and $9D) equal to zero.

Exit(A = Z.F. = 0, X = entry, Y = entry), modifies MFP.

H. Mathematical I (routines with one operand)

$EBAF: MFP = ABS(MFP)

Exit(A = entry, X = entry, Y = entry), modifies MFP.

$F09E: MFP = ATN(MFP)

Modifies $5E,$5F,$62-$66,$92-$9C,MFP,$A3,SFP,$AB,$AD,$AE.

$EED0: MFP = −MFP

Exit(X = entry, Y = entry), modifies MFP.

$EFEA: MFP = COS(MFP)

Modifies $D,$16,$5E,$5F,$62-$66,$92-$9C,MFP,$A3,SFP,$AB,$AD,$AE.

$EF09: MFP = EXP(MFP)

Modifies $D,$5E,$5F,$62-$65,$92,$98-$9C,MFP,$A3,SFP,$AB,$AD,$AE.

$EC23: MFP = INT(MFP)

Modifies $D,MFP.

$E941: MFP = LOG(MFP)

Modifies $5E,$5F,$62-$66,$92-$9C,MFP,$A3,SFP,$AB,$AD,$AE.

$DE98: MFP = NOT(MFP). This routine returns MFP = 1 if MFP = 0, else routine returns MFP = 0.

Modifies MFP, puts $11 equal to zero.

$EB90: MFP = SGN(MFP)

Exit(Y = 0), modifies MFP.

$EB82: Accumulator = SGN(MFP)

Exit(A = FF if MFP < 0, A = 0 if MFP = 0 and A = 1 if MFP > 0, X = entry, Y = entry).

$EFF1: MFP = SIN(MFP)

Modifies $D,$16,$5E,$5F,$62-$66,$92-$9C,MFP,$A3,SFP,$AB,$AD,$AE.

(continued)

Suppose one wishes to translate a BASIC subroutine to m.l. In that case the m.l. routine can be called from BASIC by means of the & symbol. The & symbol causes an unconditional jump to location $3F5 where the user can insert a JMP instruction to the start of the m.l. program.

After the execution of the & symbol, the text pointer of BASIC, which is in the locations $B8 and $B9, points to the next character of the line (spaces are ignored). Thus, if we have the line

```
10 & A1, BQ, C
```

where A1, BQ and C are reals, the text pointer points—after the execution of the & symbol—to the A. Suppose we wish to multiply A1 and BQ and store the result in C. We then first have to determine the starting location of the storage area of the value of A1 in memory. This can be done by making use of the subroutine $DFE3, listed under the heading Names in table 1. A JSR to $DFE3 in the m.l. program executes an Applesoft routine which puts the name of the variable (in this case A1) in $81 and $82; the status of the variable (in this case real) in $11 and $12; the pointer to the location of the variable in $9B and $9C; and, most important, the pointer to the value of the variable in $83 and $84, as well as in the Accumulator and the Y register.

Now that the starting location of the value of A1 is known, the value of A1 can be pulled into the MFP. For this purpose, the Copy routine $EAF9 can be used. Since the entry of this routine corresponds with the exit of $DFE3, the subroutine call to $EAF9 can be placed directly behind the subroutine call to $DFE3.

Now that we have stored A1 in the MFP, we can proceed to analyzing line 10. After the execution of subroutine $DFE3, the text pointer points to the first character behind the name of the variable, which is—in our example—a comma. If one plans to write a serious m.l. program, it might be useful to check whether there is indeed a comma behind the name.

For checking purposes, various routines are listed under the heading Charget-Charcheck. For instance, to check whether a comma is present, a JSR to $DEBE can be executed. In case the character is not a comma, the "SYNTAX ERROR" message is displayed and Applesoft gives a warm

start on BASIC. If, on the other hand, a comma is present, the text pointer is advanced and points now to the letter B.

To obtain the starting location of the storage area of the value of BQ, again a JSR $DFE3 is executed. Since AI and BQ have to be multiplied, BQ must be stored in the SFP. To accomplish this, subroutine $E9E3 is used, which also can be placed directly behind the JSR $DFE3 instruction, because the exit of $DFE3 corresponds with the entry of $E9E3. Note that it is necessary to fill the MFP before the SFP, because $AB is set when BQ is pulled in the SFP.

As can be seen, the entry of the multiplication routine $E982 corresponds with the exit of $E9E3. So after the JSR to $E9E3, the multiplication can be carried out by means of a JSR $E982. Note that the m.l. program can be reduced by several bytes by using JSR $E97F instead of the last two mentioned subroutine calls.

Finally, the result of the multiplication, which is in the MFP, has to be stored in C. Before this is done, a JSR to $DEBE is executed to check whether the text pointer points to a comma. Next, the starting location of the storage area of the value of C is determined by means of a JSR $DFE3 instruction. To store the value of C in memory, the Copy routine $EB2B can be used. Since the entry of this routine is (X,Y), whereas the exit of $DFE3 is (A,Y), the instruction TAX must be inserted before the instruction JSR $EB2B.

After the last execution of $DFE3, the text pointer points to the end of line 10, so a RTS instruction in the m.l. program returns control to the BASIC program which will restart execution at the line number following line 10. The complete m.l. program is given in example 2 (page 40).

The routine $DFE3 can also be used to find the start of the storage area of integer (%) variables, elements of arrays, and arrays. If one wishes to use matrix expressions in the & statement, it is necessary to store the hex value 40 in $14 because otherwise Applesoft will interpret the matrix names in the & statement as names of simple variables. Be sure you don't forget to put $14 back on zero before returning to BASIC, because otherwise strange things may happen.

Table 1: Applesoft Routines (continued)

$EE8D: MFP = SQR(MFP)

Modifies $D,$5E,$5F,$62-$66,$8A-$8E,$92-$9C,MFP,$A3, SFP,$AB,$AD,$AE.

$F03A: MFP = TAN(MFP)

Modifies $D,$16,$5E,$5F,$62-$66,$8A-$8E,$92-$9C,MFP, $A3,SFP,$AB,$AD,$AE.

$EFAE: MFP = RND(MFP). See Applesoft manual for argument RND function.

Modifies $5E,$5F,$62-$65,$92,MFP,SFP,$C9-$CD.

I. Mathematical II (routines with two operands)

Add

$E7C1: MFP = SFP + MFP, $AB must be determined before subroutine call.

Entry(A = Z.F. = $9D), modifies $92,MFP,SFP.

$E7BE: Pull memory (→ A,Y) in SFP, determine $AB, add: MFP = SFP + MFP.

Entry(A,Y), modifies $5E,$5F,$92,MFP,SFP,$AB.

AND

$DF55: MFP = SFP AND MFP. Routine returns MFP = 1 if MFP and SFP are both unequal to zero, else routine returns MFP = 0.

Modifies MFP, puts $11 equal to zero.

Divide

$EA69: MFP = SFP/MFP, $AB must be determined before subroutine call.

Entry(A = Z.F. = $9D), modifies $62-$66,MFP,SFP.

$EA66: Pull memory (→ A,Y) in SFP, determine $AB, divide: MFP = SFP/MFP.

Entry(A,Y), modifies $5E,$5F,$62-$66,MFP,SFP,$AB.

Multiply

$E982: MFP = SFP x MFP, $AB must be determined before subroutine call.

Entry(A = Z.F. = $9D), modifies $62-$65,MFP.

$E97F: Pull memory (→ A,Y) in SFP, determine $AB, multiply: MFP = SFP x MFP.

Entry(A,Y), modifies $5E,$5F,$62-$65,MFP,SFP,$AB.

$E2B6: Multiply two byte integer in $AD (low) and $AE (high) with two byte integer in $64 (low) and Accumulator (high). Store product in X register (low) and Y register (high).

Entry(A), Exit(X = low byte product, Y = high byte product), modifies $65,$AD,$AE, puts $99 equal to zero.

Or

$DF4F: MFP = SFP OR MFP. Routine returns MFP = 0 if MFP = SFP = 0, else routine returns MFP = 1.

Modifies MFP, puts $11 equal to zero.

Power

$EE97: MFP = SFP^{MFP}.

Entry(A = Z.F. = $9D), modifies $D,$5E,$5F,$60-$66,$8A-$8E, $92-$9C,MFP,$A3,SFP,$AB,$AE,$AD.

Subtract

$E7AA: Determine $AB, subtract: MFP = SFP – MFP.
Modifies $92,MFP,SFP,$AB.

$E7A7: Pull memory (→ A,Y) in SFP, determine $AB, subtract: MFP = SFP – MFP.
Entry(A,Y), modifies $5E,$5F,$92,MFP,SFP,$AB.

J. Names

1. Purpose

The Names routine can be used—during the evaluation of the & statement—to find the name, the status and the starting location of the storage area of simple variables, array elements and arrays.

2. Name routine

$DFE3: At the start of the execution of $DFE3, the text pointer must point to the first character of the name. After the execution of $DFE3, the text pointer points to the first character behind the name and the name and status locations are filled according to the table below.

	Name variable or array (el.)		Status variable or array (el.)	
	$81	$82	$11	$12
Real	pos	pos	0	0
String ($)	pos	neg	FF	0
Integer (%)	neg	neg	0	80

For example, if a variable has the name AB, $81 and $82 will contain the hex values 41 and 42 respectively, whereas if a variable has the name AB%, $81 and $82 will be loaded with the hex values C1 and C2. In the latter case, $12 is put equal to the hex value 80, to indicate that the variable is integer valued.

Furthermore, Applesoft loads the pointer to the start of the storage area of the variable or the array in $9B (low) and $9C (high). The pointer to the start of the storage area of the value of the variable or the array element is loaded in A=$83 (low) and Y=$84 (high). If an array element is evaluated, the pointer to the first element of the array is stored in $94 (low) and $95 (high).

In case one wishes to use matrix expressions in the & statement (for instance & A = A – B, where A and B are matrices), the hex value 40 must be stored in $14 before executing $DFE3. Before returning to BASIC, $14 has to be reset to zero again.

Under the assumption that no strings are used in the BASIC program (which may lead to house cleaning activities), the following locations may be modified by the execution of $DFE3.

1. At the evaluation of simple variable names: $10, $11, $12, $81-$84, $94-$97, $9B, $9C, $B8, $B9.
2. At the evaluation of array elements: $F, $10-$12, $81-$84, $94-$97, $9B, $9C, MFP, $AE, $AD, $B8, $B9. In addition, other locations may be modified, depending on the expressions in the subscripts.
3. At the evaluation of (already dimensioned) array names which have to be interpreted as matrix names: $10, $11, $12, $81, $82, $9B, $9C, $B8, $B9.

(continued)

Apart from using names in the & statement, one can also insert expressions. For instance, & SIN(1) + SQR(B). To evaluate such an expression, subroutine $DD67 can be executed in the m.l. program. The result of the expression is stored in the MFP. If the result has to be converted to an integer value, a JSR $E105 or a JSR $E6F8 instruction can be used instead of the JSR $DD67 instruction.

It might be possible that a wrong input to the m.l. program, or an error during the execution of the m.l. program, is detected. This will, for example, be the case if a matrix that is to be inverted turns out to be not a square matrix. In that case, one may want to let Applesoft print an error message—indicating the kind of the error—with the line number of the & statement that caused the error. For this purpose, the routines listed under the heading Errors may be used. In the case of the wrongly dimensioned matrix, a JMP $E196 instruction, for instance, displays the message "BAD SUBSCRIPT IN XX." After displaying the message Applesoft returns to BASIC command level.

Although there are more routines in table 1, it seems superfluous to discuss them here, since it will now be obvious how to use them. Instead, an example will be given to show how to integrate some of the routines in a matrix multiplication program.

A Matrix Multiplication Program

When written in BASIC, a matrix multiplication subroutine consists of the statements

```
499 REM MATRIX MULTIPLICA-
    TION : C(R,P) = A(R,S) ×
    B(S,P)
500 FOR I = 1 TO P
510 FOR J = 1 TO R
520 LET D = 0
530 FOR K = 1 TO S
540 LET D = D + A(J,K) ×
    B(K,I)
550 NEXT K
560 LET C(J,I) = D
570 NEXT J
580 NEXT I
590 RETURN
```

To execute this subroutine, the following main program can be used:

```
 10 INPUT "DIMENSIONS
    MATRICES P,R,S ? ";P,R,S
 20 DIM A(R,S),B(S,P),C(R,P)
 30 FOR I = 1 TO R
 40 FOR J = 1 TO S
 50 LET A(I,J) = I + J
 60 NEXT J
 70 NEXT I
 80 FOR I = 1 TO S
 90 FOR J = 1 TO P
100 LET B(I,J) = I × J
110 NEXT J
120 NEXT I
130 GOSUB 500
140 STOP
```

In the main program, the matrices A and B are dimensioned and initialized. Next, the matrix multiplication subroutine is called to put C equal to the product of A and B. If a m.l. program is written to multiply two matrices, the subroutine call at line 130 can be replaced by

```
130 & C = A × B
```

Although the matrix names in the & statement can be chosen freely, we will use in the sequel the names C, A and B to denote the respective matrices. The dimensions of the matrices will be denoted by the same letters as in the BASIC program (i.e. P, R and S).

The m.l. program can be split up into a main program and several subroutines. The main program performs the evaluation of the & statement. The first subroutine, called FNAME, takes care of the calculation of the pointers to the storage areas of the matrices C, A, and B in memory. The second subroutine, called MATMULT, is used for the actual matrix multiplication. Two other subroutines, ADD and ADDS, are called by FNAME and MATMULT to do some frequently occuring additions. A discussion of the functions of the various routines—which are listed in table 2— follows.

1) The main program ($4000-$4022)

The main program is written solely to control the multiplication of two matrices. Therefore it has to be replaced by another main program if the number of matrix operations is extended (with, for instance, add, subtract and inverse). The comment inserted in the listing shows how the program works.

Table 1: Applesoft Routines (continued)

K. Normalize

$E82E: Normalize MFP
Exit(Y = 0), modifies MFP.

L. Pack

$EB72: Pack extension byte in MFP.
Exit(X = entry, Y = entry), modifies MFP.

Example 2

$3F5 : JMP $5000		Jump to multiplication program
$5000 : JSR $DFE3		Find starting location of value of first variable
$5003 : JSR $EAF9		Pull first variable into the MFP
$5006 : JSR $DEBE		Check on comma in & statement
$5009 : JSR $DFE3		Find starting location of value of second variable
$500C : JSR $E9E3	JSR $E97F	Pull second variable in the SFP, and
$500F : JSR $E982		Multiply MFP with SFP, store product in MFP
$5012 : JSR $DEBE		Check on comma
$5015 : JSR $DFE3		Find starting location of value of third variable
$5018 : TAX		Prepare entry store routine
$5019 : JSR $EB2B		Store product in third variable
$501C : RTS		Return to BASIC

Table 2: Listings of Machine Language Programs

A. The main program

Purpose

The evaluation of the & statement: & C = A × B, where C,A and B are matrices.

Listing	Comment
$3F5 : JMP $4000	Init jump location for & statement.
$4000 : LDX #$F8 $4002 : STX $08	Init location $08 for FNAME.
$4004 : JSR $4025	Execute FNAME on first matrix (C).
$4007 : LDA #$D0 $4009 : JSR $DEC0	Check on "=" in & statement.
$400C : JSR $4025	Execute FNAME on second matrix (A).
$400F : LDA #$CA $4011 : JSR $DEC0	Check on "×" in & statement.
$4014 : JSR $4025	Execute FNAME on third matrix (B).
$4017 : LDA $06 $4019 : STA $71 $401B : LDA $07 $401D : STA $72	Restore column length of A (= column length of C) in $71 and $72.
$401F : JSR $40A5	Execute MATMULT.
$4022 : RTS	Return to BASIC.

B. Subroutine FNAME

Purpose

Find name of array, check whether array has two dimensions, each less than 256. Store dimensions in $FC,X (second dimension) and $FD,X (first dimension). Calculate column length of array (in bytes) and store it in $71 (low) and $72 (high). Calculate pointer to storage area of first element of second column of array, and store pointer in $0,X + 2 and $1,X + 2. FNAME can be called successively three times (or less). Before the first call, the hex value F8 must be stored in location $08. At the start of FNAME, the X register is loaded with the value in location $08. During the execution of FNAME the X register is incremented by two and stored in location $08 so that the contents of location $08 are incremented by two each time FNAME is called.

(continued)

2) Subroutine FNAME ($4025-$4083)

Contrary to the main program, FNAME is constructed in such a way that it can be used for other matrix operations too. The main purpose of FNAME is to calculate the pointer to the first element of the second column of the array being evaluated. The second column is taken because it is customary to use—when working with matrices—non zero values of the subscripts only, whereas Applesoft reserves—when it encounters a DIM X(P,R) statement—P + 1 rows and R + 1 columns for the array because it allows zero subscripts. As an example, suppose that a DIM X(2,3) instruction is executed in a BASIC program. Applesoft stores the X-array columnwise (example 3, page 46).

When multiplying the X matrix with another matrix, only the underscored elements have to be taken into account. Since the first column contains no underscored elements, it can be skipped.

Subroutine FNAME can be called successively (at most) three times. At the first call, location $08 must contain the hex value F8, being the start of the storage area, plus 4 of the matrix information. Consulting the memory map of FNAME in table 2, it can be seen that the dimensions of the C array, P + 1 and R + 1, are stored in $F4 and $F5 and the pointer to the first element of the second column of the C array (i.e., the pointer to C(0,1) in $FA and $FB. Since location $08 is automatically incremented by 2, each time FNAME is called, the dimensions of the next array (i.e., the A array) will—at the second call of FNAME—be stored in $F6 and $F7 and the pointer in $FC and $FD. The information of the B array is stored in $F8, $F9, $FE and $FF.

Apart from the calculation of the pointer, FNAME also checks whether the array being evaluated has two dimensions, and whether the size of each dimension is less than 256. The latter check is necessary because MATMULT can handle matrices with dimensions less than 255 only, which will be sufficient for almost all practical purposes.

Finally, at each call of FNAME, the column length of the array being evaluated (which equals 5 times the number of column elements, since reals use 5 bytes of memory) is calculated and stored in $71 and $72.

Table 2: **Listings of Machine Language Programs (continued)**

Memory Map of FNAME

$06, $07	Column length (in bytes) of first array (C).
$08	Pointer to storage area of array information.
$71, $72	Column length (in bytes) of third array (B).
$F4	Second dimension of first array (C).
$F5	First dimension of first array (C).
$F6, $F7	Idem for second array (A).
$F8, $F9	Idem for third array (B)
$FA, $FB	Pointer to first element of second column of first array (C).
$FC, $FD	Idem for second array (A).
$FE, $FF	Idem for third array (B).

Listing FNAME	Comment
$4025 : LDA #$40	Put $14 equal to 40 for search of matrix name.
$4027 : STA $14	
$4029 : JSR $DFE3	Determine pointer to start storage area of array.
$402C : LDX $08	X is loaded with pointer to storage area array information.
$402E : LDA $12	Check whether array contains reals.
$4030 : ORA $11	
$4032 : BEQ $4037	
$4034 : JMP $DD76	If not, display "TYPE MISMATCH".
$4037 : STA $14	Put $14 back on zero.
$4039 : LDY #$04	
$403B : LDA #$02	
$403D : CMP ($9B),Y	Compare number of dimensions of array with 2.
$403F : BEQ $4044	
$4041 : JMP $E196	If not equal, display "BAD SUBSCRIPT".
$4044 : INY	
$4045 : LDA($9B),Y	Check whether dimensions of array are both less than 256. If not, display "BAD SUBSCRIPT". If yes, store second dimension in $FC,X and first dimension in $FD,X.
$4047 : BNE $4041	
$4049 : INY	
$404A : LDA($9B),Y	
$404C : STA $FC,X	
$404E : INX	Note that at the first call of FNAME, the X register is loaded with F8. The dimensions of the first array are thus stored in $F4 and $F5.
$404F : INY	
$4050 : CPY #$09	
$4052 : BNE $4045	
$4054 : TYA	Accumulator contains 9 here.
$4055 : CLC	To obtain the pointer to the storage area of the first element in the array, 9 is added to the pointer in $9B and $9C. The result is stored in $00,X (low) and $01,X (high). At the first call of FNAME X equals FA here, so the pointer is stored in $FA and $FB.
$4056 : ADC $9B	
$4058 : STA $00,X	
$405A : LDA $9C	
$405C : ADC #$00	
$405E : STA $01,X	
$4060 : LDA $FB,X	Calculate the column length of the array by multiplying the size of the first dimension (which was stored in $FB,X) with 5. The column length is stored in $71 and $72.
$4062 : LDY #$00	
$4064 : STY $72	
$4066 : ASL	
$4067 : ROL $72	
$4069 : ASL	
$406A : ROL $72	
$406C : ADC $FB,X	
$406E : STA $71	
$4070 : BCC $4074	
$4072 : INC $72	

```
$4074 : CPX #$FA      Is it the first call of FNAME?
$4076 : BNE $407E
$4078 : STA $06       If yes, save column length of the array in $06 and
$407A : LDY $72       $07.
$407C : STY $07
$407E : JSR $4087     Add column length to the last calculated pointer
$4081 : STX $08       to obtain the pointer to the storage area of the
$4083 : RTS           first element of the second column of the array.
                      Store X in $08 for next calls of FNAME and
                      return to main program.
```

C. **Subroutines ADD and ADD5**

Purpose ADD

Add two byte integer in $71 (low) and $72 (high) to two byte integer in $00,X (low) and $01,X (high). Store result in $00,X (low) and $01,X (high).

Entry(X), Exit(A = $01,X,X = entry,Y = entry).

Purpose ADD5

ADD 5 to two byte integer in $00,X (low) and $01,X (high). Store result in $00,X (low) and $01,X (high).

Entry(X), Exit(A = $00,X,X = entry,Y = entry).

```
Listing ADD              Listing ADD5
$4085 : LDA $71          $4095 : CLC
$4087 : CLC              $4096 : LDA $00,X
$4088 : ADC $00,X        $4098 : ADC #$05
$408A : STA $00,X        $409A : STA $00,X
$408C : LDA $72          $409C : BCC $40A0
$408E : ADC $01,X        $409E : INC $01,X
$4090 : STA $01,X        $40A0 : RTS
$4092 : RTS
```

D. **Subroutine MATMULT**

Purpose

Multiply matrix A(R,S)	(dimensions in $F6 (S+1) and $F7 (r+1), pointer in $FC and $FD)
with matrix B(S,P)	(dimensions in $F8 (P+1) and $F9 (S+1), pointer in $FE and $FF)
and store result in matrix C(R,P)	(dimensions in $F4 (P+1) and $F5 (R+1), pointer in $FA and $FB)

where P,R, and S each have to be less than 255.

Memory map of MATMULT

$06	cr : Row counter for C.
$07	cs : Multiplication counter for row/column multiplication.
$17, $18	hp_A : pointer to first element of current row of A.
$19, $1A	p_B : pointer to first element of current column of B.
$71, $72	k = 5(R+1) : Column length of A (in memory).
$F4	P + 1 at entry. Used as column counter for C.
$F5	R + 1 = number of elements per column of C (in memory)
$F6	S + 1 : S equals the number of multiplications necessary to multiply a row of A with a column of B.
$F8, $F9	pc_A : pointer to current element of A.
$FA, $FB	pc_C : pointer to current element of C.
$FC, $FD	p_A : pointer to first element of second column of A.
$FE, $FF	pc_B : pointer to current element of B.

(continued)

The column length of the C array, which equals the column length of the A array, is saved in locations $06 and $07, because the latter column length is needed later for MATMULT.

3) Subroutine MATMULT ($40A5-$4124)

Before the matrices are multiplied, the dimensions are checked to determine whether they satisfy the conditions for multiplication. Next, the multiplication is carried out as indicated by the flow diagram in figure 1. The flow diagram shows that the pth column of C (p = 2,..P+1) is obtained by multiplying the R rows of A (i.e., row 2,...R+1) each with the pth column of B (p = 2,...P+1). Note that at a row/column multiplication, the product of the first element of a row and the first element of a column is omitted since these elements have zero subscripts. The elements of the rows of A are separated by a distance of k (= 5(R+1)) bytes from each other in memory, so that each time a next row element of A is needed, k has to be added to pc_A. After a row of A, not being the last row, has been multiplied with a column of B, the hp_A pointer is incremented by 5 and pc_A is put equal to hp_A, so that pc_A now points to the second element of the next row of A. If a column of C, not being the last column, has been filled, hp_A is put equal to its starting value. That is, p_A, and p_B is put equal to pc_B, which at that time points to the first element of the next column of B.

The flow diagram further shows how the multiplication of a row with a column is performed. The stack is used to store the sum of the products obtained so far, and each time a row element is multiplied with a column element, the stack is pulled into the SFP and the newly-obtained product is added to it. The result is then pushed on the stack again. This process is continued until the row/column multiplication is ready. The row/column product is then stored in memory.

Note that the subroutine address pushed on the stack at locations $40CE-$40D3 is the address minus one of the instruction following the JMP instruction at location $40E5. If the MATMULT subroutine is relocated to another part of memory, this subroutine address must be adjusted.

Table 2: Listings of Machine Language Programs (continued)

Listing MATMULT	Comment
$40A5 : LDA $F4	Check dimensions for multiplication. If an error is detected, display "BAD SUBSCRIPT".
$40A7 : CMP $F8	
$40A9 : BEQ $40AE	
$40AB : JMP $E196	
$40AE : LDA $F5	
$40B0 : CMP $F7	
$40B2 : BNE $40AB	
$40B4 : LDA $F6	
$40B6 : CMP $F9	
$40B8 : BNE $40AB	
$40BA : DEC $F4	P = P − 1 : decrement column counter for C.
$40BC : BNE $40BF	If P equals zero, matrix multiplication is ready
$40BE : RTS	Return to main program.
$40BF : LDA $F5	cr = R + 1 : init row counter for C.
$40C1 : STA $06	
$40C3 : LDX #$03	$hp_A = p_A$
$40C5 : LDA $FC,X	$p_B = cp_B$
$40C7 : STA $17,X	
$40C9 : DEX	
$40CA : BPL $40C5	
$40CC : BMI $4100	Always taken.
$40CE : LDA #$40	Push subroutine address for routine $DE47 on stack.
$40D0 : PHA	
$40D1 : LDA #$E7	
$40D3 : PHA	
$40D4 : JSR $DE10	Push MFP on stack.
$40D7 : LDA $F8	Load (pc_A) in MFP.
$40D9 : LDY $F9	
$40DB : JSR $EAF9	
$40DE : LDA $FE	Load (pc_B) in SFP and
$40E0 : LDY $FF	
$40E2 : JSR $E97F	Multiply MFP with SFP. Store product in MFP.
$40E5 : JMP $DE47	Pull stack into SFP.
$40E8 : JSR $E7C1	Add MFP and SFP. Store sum in MFP.
$40EB : LDX #$F8	$pc_A = pc_A + k$
$40ED : JSR $4085	
$40F0 : LDX #$FE	$pc_B = pc_B + 5$
$40F2 : JSR $4095	
$40F5 : DEC $07	cs = cs − 1
$40F7 : BNE $40CE	If cs equals zero, row/column product is ready.
$40F9 : LDX $FA	Store MFP in (pc_C).
$40FB : LDY $FB	
$40FD : JSR $EB2B	
$4100 : LDX #$FA	$pc_C = pc_C + 5$
$4102 : JSR $4095	
$4105 : DEC $06	cr = cr − 1
$4107 : BEQ $40BA	Is column of C filled? If yes, init hp_A, p_B and cr.
$4109 : LDX #$17	$hp_A = hp_A + 5$
$410B : JSR $4095	
$410E : STA $F8	$pc_A = hp_A$
$4110 : LDA $18	
$4112 : STA $F9	
$4114 : LDA $19	$pc_B = p_B$: restore column counter for B.
$4116 : STA $FE	
$4118 : LDA $1A	
$411A : STA $FF	
$411C : LDA $F6	cs = S + 1 : init multiplication counter.
$411E : STA $07	
$4120 : JSR $E84E	Initialize MFP to zero.
$4123 : BEQ $40F0	Always taken.

Some Final Remarks

Probably not all Apple owners will have Applesoft in ROM. However, since the disk and tape versions of Applesoft which I have seen do not differ by more than a few bytes from the ROM version, it will be no big problem to convert the entries of the routines listed in table 1 to these versions. The easiest way to do this is to find someone who has Applesoft in ROM, so that the differences can readily be traced back by comparing the versions with each other. In case no ROM version is available, one can use the subroutine entry locations, which are found at the beginning of the Applesoft program. The sequence of the first 64 subroutine entry locations* corresponds with the listing of the tokens in the Applesoft manual (#A2L0006 on page 121). Next, the entry locations of the routines for SIGN to MID$ follow. The rest of the entry locations* are for +, −, ×, /, Ο, AND, OR, unary minus, NOT and comparison. Before each of the latter entries, a code, indicating the order of the operation, is inserted.

Looking at table 3, where the entries of the routines for the ROM version are listed alphabetically, with the entries found in tape or disk versions of Applesoft, it will become apparent what differences there are. After that, the entry locations of the routines in table 1 can be converted accordingly.

As a last point I wish to express my admiration for the ingenuity of the writers of Applesoft. During my study of Applesoft, I often searched for hours for what was happening, in a seemingly endless sequence of (recursive) subroutines, which taught me a lot about m.l. programming. Apart from a few errors that were made (for instance, a zero byte forgotten between $E101 and $E102, so that the program

```
10 A% = -32768.00049 :
   PRINT A%
```

gives a surprising result) I came to the conclusion that Applesoft is a very good interpreter.

I also wish to express my gratitude to Mr. F. Curvers of the Erasmus University in Rotterdam, for providing me with an excellent cross reference of Applesoft, without which my work would have been far more difficult.

*Add one to the location found in the listing because the subroutines are executed via the RTS instruction.

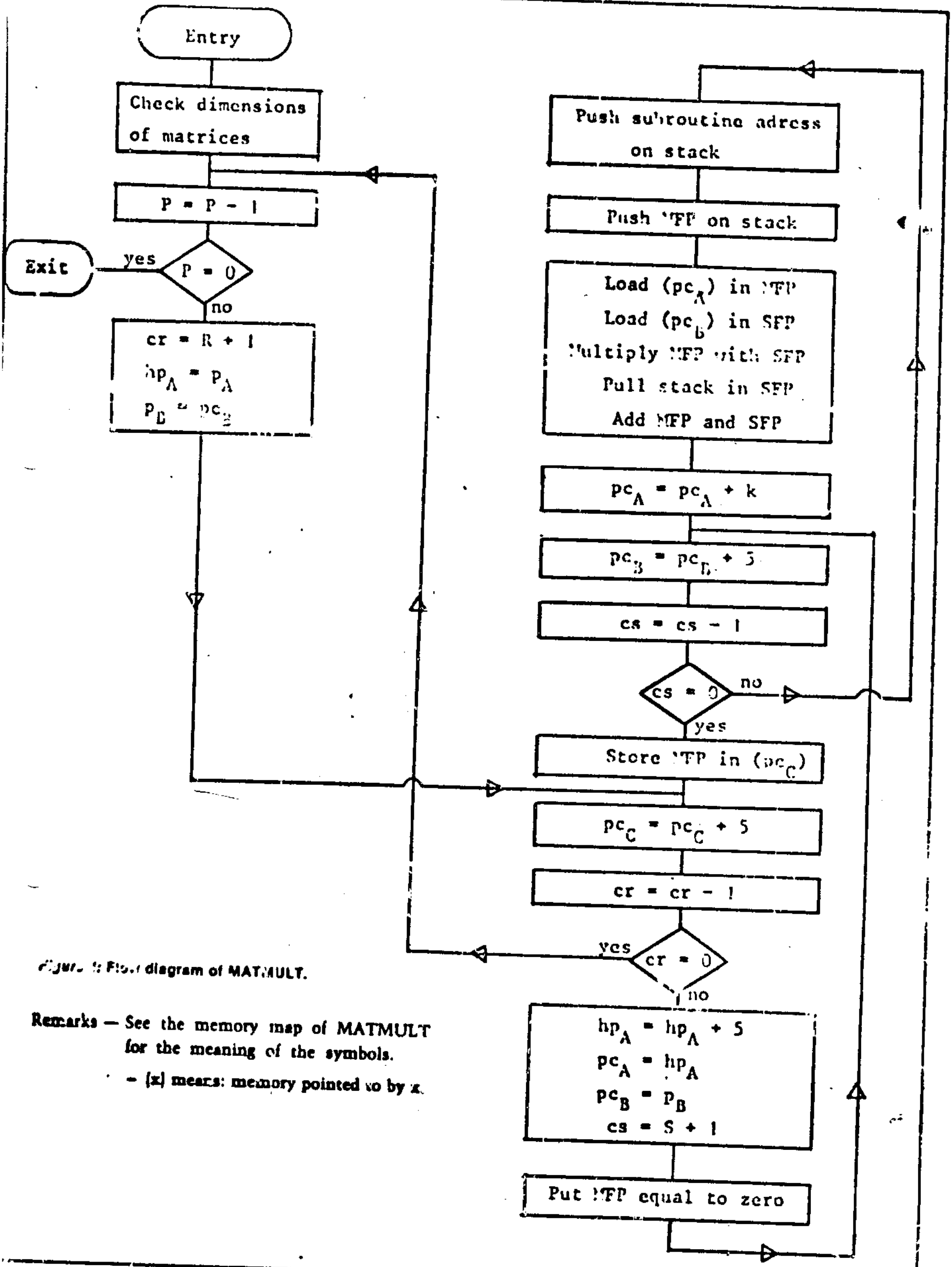

Figure [illegible]: Flow diagram of MATMULT.

Remarks — See the memory map of MATMULT for the meaning of the symbols.
- (x) means: memory pointed to by x.

Cornelis Bongers is an assistant professor of statistics at the Erasmus University in Rotterdam. He uses his Apple II for solving statistical problems (for instance, likelihood maximalization). Another important field of application is finding the solution of standardization problems, which in essence means: finding the set of sizes that minimizes the overall costs caused by the standardization of a product. As a hobby, he develops utility programs for the Apple, such as an assembler cross reference program and a disk-to-tape dump utility.

Example 8

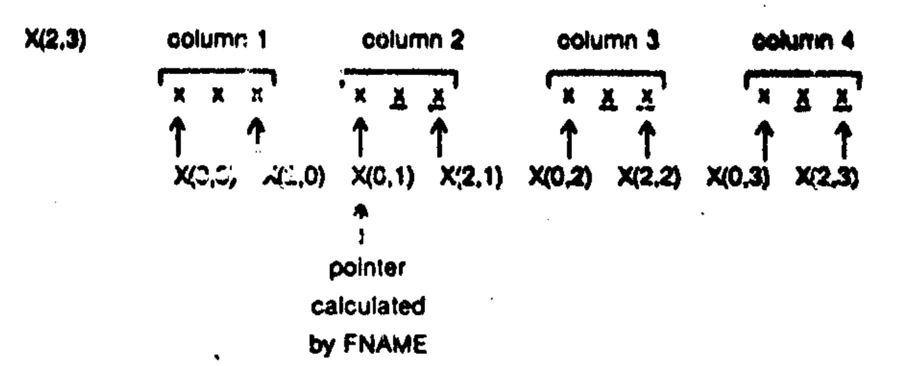

Table 3: Applesoft (ROM) Entry Locations

Entry	Dec token	Key-word	Entry	Dec token	Key-word	Entry	Dec token	Key-word
$3F5	175	&	$F3D8	144	HGR2	$F3BC	167	RECALL
$E982	202	×	$F286	163	HIMEM:	$D9DC	178	REM
$E7C1	200	+	$F232	142	HLIN	$D849	174	RESTORE
$E7AA	201	–	$FC58	151	HOME	$F318	166	RESUME
$EA69	203	/	$F6FE	147	HPLOT	$D96B	177	RETURN
	209	<	$F7E7	150	HTAB	$E686	233	RIGHT$
$DF65	208	=	$D9C9	173	IF	$EFAE	219	RND
	207	>	$F1DE	139	IN#	$F721	152	ROT=
$EBAF	212	ABS	$DBB2	132	INPUT	$D912	172	RUN
$DF55	205	AND	$EC23	211	INT	$D8B0	183	SAVE
$E6E5	230	ASC	$F277	158	INVERSE	$F727	153	SCALE=
—	197	AT	$E65A	232	LEFT$	$DEF9	215	SCRN(
$F09E	225	ATN	$E6D6	227	LEN	$EB90	210	SGN
$F1D5	140	CALL	$DA46	170	LET	$F775	154	SHLOAD
$E646	231	CHR$	$D6A5	188	LIST	$EFF1	223	SIN
$D66A	189	CLEAR	$D8C9	182	LOAD	$DB16	195	SPC(
$F24F	160	COLOR=	$E941	220	LOG	$F262	169	SPEED=
$D896	187	CONT	$F2A6	164	LOMEM:	$EE8D	218	SQR
$EFEA	222	COS	$E691	234	MID$	—	199	STEP
$D995	131	DATA	$D649	191	NEW	$D86E	179	STOP
$E313	184	DEF	$DCF9	130	NEXT	$F39F	168	STORE
$F331	133	DEL	$F273	157	NORMAL	$E3C5	228	STR$
$DFD9	134	DIM	$DE98	198	NOT	$DB16	192	TAB(
$F769	148	DRAW	$F26F	156	NOTRACE	$F03A	224	TAN
$D870	128	END	$D9EC	180	ON	$F399	137	TEXT
$EF09	221	EXP	$F2CB	165	ONERR	—	196	THEN
$F280	159	FLASH	$DF4F	206	OR	—	193	TO
$E354	194	FN	$DFCD	216	PDL	$F26D	155	TRACE
$D766	129	FOR	$E764	226	PEEK	$A	213	USR
$E2DE	214	FRE	$F225	141	PLOT	$E707	229	VAL
$DBA0	190	GET	$E77B	185	POKE	$F241	143	VLIN
$D921	176	GOSUB	$D96B	161	POP	$F256	162	VTAB
$D935	171	GOTO	$E2FF	217	POS	$E784	181	WAIT
$F39C	136	GR	$F1E5	138	PR#	$F76F	149	XDRAW
$F6E9	146	HCOLOR=	$DAD5	186	PRINT	$EE97	204	^
$F3E2	145	HGR	$DBE2	135	READ			